Description

This checklist must be accurately filled out by a qualified engineer prior to releasing a new QuickUSB driver for distribution to end-users. Once the checklist has been completed and approved by management, the software may be released for deployment through the approved distribution methods.

# Overview

## Flow of Execution

* When the driver is associated to a hardware device by Windows, the first method that is called is DriverEntry(). That function is responsible for creating and registering the driver with WDF.
* After the driver has been created, the EvtDeviceAdd() method is called for each new hardware device that is to be controlled through this driver. That function is responsible for creating IO Request queues (to store IO Requests made to the device) and to register one or more device interfaces (so that a user-mode application may access the device).
* Once a device has been added, the EvtDevicePrepareHardware() method is called to prepare the device for use. This function will select the USB interface to use and set the power policy for the device.
* As read, write, and ioctl requests are requested to a device from user-mode, those requests get queued in the IO Request queues that were created for the device. To fulfill the requests, the WDF calls the EvtIoRead(), EvtIoWrite(), and EvtIoDeviceControl() functions to perform the requested operation. Those functions must either synchronously perform the requested operation and complete then request or asynchronously schedule the request.
* When an IO Request is perform asynchronously, a callback function and timeout is registered with the request so that once the request either completes or times out it is completed. All requests must be completed (either with or without error) or the user-mode application will deadlock.
* All requests that are not able to be completed immediately, either because they require access to hardware or perform a possibly slow operation, should be done asynchronously. Note that performing an asynchronous request in the driver does not prevent user-mode applications from issuing synchronous requests as the Windows IO manager bridges the kernel-mode user-mode gap.
* When the driver is being removed from the kernel, the EvtDriverContextCleanup() method is called to perform any necessary cleanup. Currently this function is empty as nearly all WDF resources are managed.

## IO Requests

For every IO Request issued on a device, WDF creates an IO Request Object, fills out information about the request, queues the request, then (depending on how the queue is configured) calls the appropriate IO Handler for the request type. IO Request Handlers may complete the request, forward the request, or break the request into multiple requests. The QuickUSB driver uses all of these methods to perform IO.

IO reads and writes, for example, break the IO Request, which could be for as much as 16 MB, into sub-requests that are 256 KB in size or smaller and performs this asynchronously. This design allows for maximal USB bandwidth and even provides for a performance increase over the WDM driver for the synchronous QuickUsbReadData() and QuickUsbWriteData() functions by dividing and asynchronously performing the request in pieces, substantially reducing overhead latencies.

## Contexts

Many WDF types may have associated contexts. Contexts are user-definable memory to store state information. The QuickUSB WDF driver defines contexts for each device, each IO Requests, and work items. All contexts are defined in “contexts.h”.

## Synchronization

WDF is capable of automatically handling the synchronization request dispatching and completion routine callbacks. Dispatching synchronization is defined when creating IO Queues and callback synchronization (called Synchronization Scope) is also defined when creating IO queues.

## Error handling

Errors in the driver propagate back to the QuickUSB Library, to be retrieved via the GetLastError() function, by completing requests with a status code. The QuickUSB-defined error codes are defined in “driver.h”. When a failure occurs within the driver while trying to process an IO Request, the NTSTATUS error is translated to the QuickUSB driver error equivalent via the GetDriverErrorFromNtStatus() function and stored in the device context. The error condition is detected in the library and the library requests the last error from the driver, thus returning the QuickUSB driver error code.

There is a lot of room for improvement in error reporting, but since the QuickUSB Library currently only supports a relatively simple QuickUsbGetLastError() function, the current implementation will suffice. In the future a function which is able to extract more meaningful error information and present it to the user would greatly enhance debugging.

# Driver Instantiation and Removal

## Overview

When the driver is initially associated to a hardware device by Windows (through an INI file), the first method that is called is DriverEntry(). That function is responsible for creating and registering the driver with WDF and must return successfully in order for Windows to load the driver.

## Functions (Found in driver.c)

### DriverEntry

Called on driver instantiation.

### EvtDriverContextCleanup

Called on driver removal. Currently empty.

# Device Instantiation

## Overview

After the driver has been created, the EvtDeviceAdd() method is. That function is responsible for creating IO Request queues (to store IO Requests made to the device) and to register one or more device interfaces (so that a user-mode application may access the device).

Once a device has been added, the EvtDevicePrepareHardware() method is called to prepare the device for use. This function will select the USB interface to use and set the power policy for the device.

## Functions (Found in device.c)

### EvtDeviceAdd

Called for each new hardware device that is to be controlled through this driver.

### EvtDevicePrepareHardware

Called to prepare the device for use.

### SelectInterfaces

Called by EvtDevicePrepareHardware() to select the USB descriptor interface to use for the device.

### ResetPipe

Reset the indicated USB pipe.

### StopAllPipes

Stop all IO Requests on all USB pipes.

### StartAllPipes

Start IO Request processing on all USB pipes.

### ResetDevice

Issue a USB reset on the USB device.

### AbortPipe

Abort IO Requests on the indicated USB pipe.

### AbortPipes

Abort IO Requests on all USB pipes.

# Power Management

## Overview

Power management is tightly integrated with WDF, though not yet utilized in QuickUSB. Many of the required power control methods have been stubbed out in the driver for future development.

## Functions (Found in power.c)

### EvtDeviceD0Entry

Called when the device enters the working state.

### EvtDeviceD0Exit

Called when the device exits the working state.

### SetPowerPolicy

Sets the power policy, i.e. when the driver should put the device into the idle and wake states. Though implemented, the QuickUSB firmware currently does not support power management and therefore is always in the working state (D0).

# Control Transfers

## Overview

All USB Control Transfers to QuickUSB use the IOCTL interface (via the DeviceIoControl() function). Valid IOCTL codes are defined in “ioctl.h”.

## Functions (Found in ioctl.c and control.c)

### EvtIoDeviceControl

The IOCTL request handler routine.

### EvtControlRequestCompletion

The USB control request completion routine for IOCTL requests that need to perform asynchronous control requests.

### IssueInternalVendorRequest

Creates and dispatches a QuickUSB vendor request from within the driver. This is used from within the EvtIoRead() function to send the number of bytes to read to the QuickUSB module.

### EvtInternalVendorRequestCompletion

The IO request completion routine for internally issued vendor requests.

# Bulk Transfers

## Overview

The WDF driver introduces a new, more efficient way to perform bulk requests. Previously, the WDM driver performed bulk requests via the IOCTL interface. This placed the burden of bulk transfer work in the QuickUSB Library. To perform a read request the library would first need to issue an IOCTL request to send a command transfer to QuickUSB indicating the number of bytes to read. Then, the library would issue a read request through the IOCTL interface. Additionally, the library must manually break large requests into smaller requests that the driver is able to perform. If you wanted to perform an asynchronous read you would have to issue an asynchronous read request, also through the IOCTL, placing the majority of asynchronous work and housekeeping in the driver and library. The WDF driver alleviates these issues by fully handling data requests, keeping all physical requirements (such as the 256 KB USB request limit) internal to the driver.

## IOCTL Interface

For backwards compatibility, the asynchronous IOCTLs have been implemented though their use is deprecated.

## ReadFile() and WriteFile() Interface

The WDF driver performs all bulk read and writes asynchronously and automatically handles dividing large requests into smaller ones such that library never needs to do more than issue a request for data using the standard Windows ReadFile() and WriteFile() functions. Since the Windows IO Manager automatically handles asynchronous IO, the asynchronous Windows API (use of the OVERLAPPED parameter) is inherently supported. The required control request to send the number of bytes to read for a bulk IN transfer is implemented within the driver to improve performance and abstract the requirements of hardware from the library API. Now, even within the context of the QuickUSB API, the complexity of data transfers, both synchronous and asynchronous, are handled either by Windows or from within the driver.

## Functions (Found in bulkrwr.c)

### EvtIoRead

This is the IO Request handler for read requests (usually made from ReadFile()).

### EvtIoWrite

This is the IO Request handler for write requests (usually made from WriteFile()).

### EvtIoStop

This is the IO Request handler for stop requests (usually made from CancelIo() or from the Windows IO Manager).

### PerformBulkRequest

This is the heart of bulk transfers. To maximize perform and to interface well with the Windows IO Manager all bulk transfers are performed asynchronously in kernel land (the Windows IO Manager handles making the transfers either synchronous or asynchronous in user land). This function handles both read and write data transfers, and breaks transfers into 256 KB chunks. The function creates the request objects, initializes the requests, and finally dispatches the requests. If the request is for a read transfer, the function issues a vendor command to send the number of bytes to read to the QuickUSB Module before issuing the read requests.

### EvtRequestReadWriteCompletion

This is the completion routine for the IO Requests dispatched from the PerformBulkRequest() function. Since IO transfers are broken into chucks, the routine may not simply complete the initial IO bulk request. It instead must check to see if all the split requests have finished, keeping track of any errors that occurred along the way, then complete the initial IO Request.

# Miscellaneous

## Resources

### QuickUSB.rc and resource.h

The QuickUSB.rc file contains the version information that is embedded into the output SYS file. It contains the driver name, version, and copyright information which needs to be updated with every new release of the driver.

### QuickUSB.inx

The QuickUSB.inx file is a INF template file that is used to generate the final INF output file for the project. It contains a few symbols which are expanded during INF file generation, including one that describes the KMDF co-installer version required by the driver and one for the Windows signature reference under the ‘[Version]’ section in the INX/INF.

## Sources

The sources file is essentially the build file for the driver project. It defines the driver name, the target INF file to create, external libraries to link against, and the project source files, along with a few other things. It is important to note that if a source file is either added or removed from the Visual Studio project, you must still manually edit the sources file as Visual Studio does not maintain that file.

## Functions (Found in util.c)

### GetDriverErrorFromNtStatus

This function is used in error handling to translate NTSTATUS error codes into QuickUSB driver-defined error codes that the QuickUSB Library is able to interpret.

### DbgGetVendorStr

This function returns the string representation of valid QuickUSB vendor commands. This function is only ever called for debug builds of the driver and aids in the debugging process by displaying a human-readable strings in place of a numeric value for vendor commands.

### DbgDevicePowerString

This function returns the string representation of the valid WDF power states. This function is only ever called for debug builds of the driver and aids in the debugging process by displaying a human-readable string in place of a numeric value for power-state transitions.

### QueueResetPipePassiveLevelCallback

The ResetPipe() function must be called PASSIVE\_LEVEL, but usually the functions that need to call ResetPipe() are running at DISPATCH\_LEVEL. Therefore, a work item must be created to schedule the callt o ResetPipe() to occur at PASSIVE\_LVEL. This function creates and queues a work item to reset a USB pipe.

### ResetPipeWorkItem

The worker function for resetting a pipe with a work item.

# Notes

* All synchronous functions must run at PASSIVE\_LEVEL.
* WorkItems must be used to call a function that needs to run at PASSIVE\_LEVEL from a function running at DISPATCH\_LEVEL.
* SpinLocks raise the execution level to DISPATCH\_LEVEL.
* If a function is pagable, it must run at PASSIVE\_LEVEL.
* At PASSIVE\_LEVEL, the WDF uses a FAST\_MUTEX instead of a SpinLock for synchronization.